Microservices

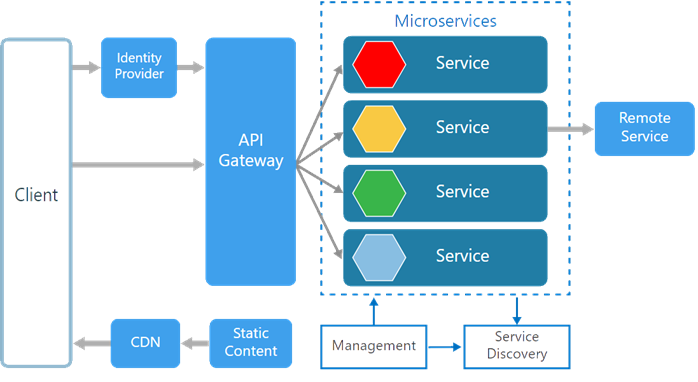
Microservices architecture is a flavor of service oriented architecture

In Microservices we decompose a big application into multiple small services.

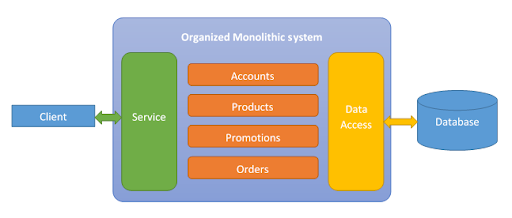
A microservice is an approach to create small services, each running in their own space that can communicate each other.

Microservices are independent services directly calling their own database.

## Microservices Architecture



Monolithic System



Monolithic System Limitations

In Monolithic System If any single application function or component fails, then the entire application goes down. Imagine a web application with separate functions including payment, login, and history. If a particular function starts consuming more processing power, the entire application’s performance will be compromised.

Monolithic architecture impacts both the development and application deployment stage.

developers cannot work independently to develop or deploy their own modules and must remain totally dependent on others, increasing overall development time.

In monolithic architecture, the database remains the same for all the functionalities even if an approach of service-oriented architecture is followed, whereas in microservices each service will have their own database.

WHY SHOULD WE USE MICROSERVICES INSTEAD OF A MONOLITHIC APPROACH?

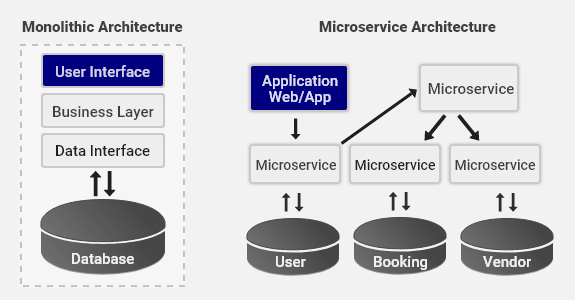
Microservices is an approach to develop small services that each run in its own process. We should develop microservices instead of one service (a monolithic approach) for a multitude of benefits, including:

* Microservices are smaller in size
* Microservices are easier to develop, deploy, and debug, because a fix only needs to be deployed onto the microservice with the bug, instead of across the board
* Microservices can be scaled quickly and can be reused among different projects
* Microservices work well with containers like Docker
* Microservices are independent of each other, meaning that if one of the microservices goes down, there is little risk of the full application shutting down

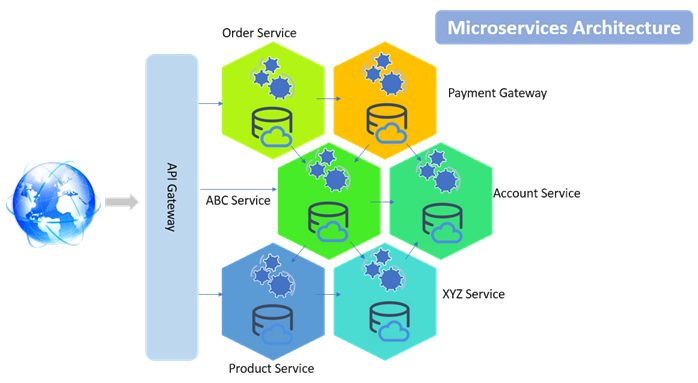
## Advantages Microservices Architecture

* Each service is small and independent.
* Services can be written in different programming language , databases, and tools for each service
* Independently develop, deploy, redeploy, version and scale component services in seconds without compromising the integrity of an application
* Better fault isolation keeps other services to work even though on got failed.
* Zero downtime upgrades.
* Services can be of from different servers or even different datacenters.
* Interaction with other services in a well-defined protocol
* Monitor, capture, and report health diagnostics
* Reliable and self-healing
* Supports continuous integration and delivery
* Easy to transfer knowledge to the new team member
* Easy to integrate with third parties

## Monolithic vs Microservices Architecture



The following is the diagrammatic representation of microservices architecture.



# Microservices and the 12 Factor App

A collection of guidelines to help to build good microservice-based applications is the Twelve-Factor app. It is a collection of 12 tips for the development of horizontally scalable applications.

The 12 factors app is a methodology to build applications the are suited for modern architectures(e.g cloud and containers and that operates well in DevOps operating environments)

Why 12 factors App

12 factors app leads to applications that are portable, high scalable and easily automatable.

Factor 1: Codebase

One single code base many deployments

Store your app code base in a version control syst em(e.g. git),dedicate one repository for one app,create many deployments out of that repository.

Build on top of one codebase, fully tracked by a Version Control System (VCS). Deployments should be automatic, so everything can run in different environments without work. You should always have one repository for an individual application to ease CI/CD pipelines.

Factor 2: Dependencies

Do not copy any dependencies to the project codebase, instead use a package manager. Always remember to use the correct versions of dependencies so that all environments are in sync and reproduce the same behavior.

Factor 3: Config

Store the config in Environment Variable. There should be a strict separation between config and code. The code should remain the same irrespective of where the application is being deployed, but configurations can vary.

Factor 4: Backing Services

Treat backing services as attached resources as your services should be easily interchangeable. You must be able to easily swap the backing service from one provider to another without code changes. This will ensure good portability and help maintain your system.

Factor 5: Build, Run, Release

A twelve-factor application requires a strict separation between Build, Release and Run stages. Every release should always have a unique release ID and releases should allow rollback. Automation and maintaining the system should be as easy as possible. Then you put everything together in something that can be released and installed in the environment and then be able to run it.

Factor 6: Stateless Processes

You should not be introducing state into your services, applications should execute as a single, stateless process. The Twelve-factor processes are stateless and share-nothing. This factor lies at the core of microservices architecture.

Factor 7: Port Binding

Your service should be visible to others via port binding. If you built a service, make sure that other services can treat this as a resource if they wish. The twelve-factor app is completely self-contained.

Factor 8: Concurrency

Break your app into much smaller pieces rather than trying to make your application larger (by running a single instance on the most powerful machine available). Small, defined apps allow scaling out as needed to handle the varying loads. Each process should be individually scaled, with Factor 6 (Stateless), it is easy to scale the services.

Factor 9: Disposability

Processes should be less time-consuming. Make sure you can run and stop fast. And that you can handle failure. Without this, automatic scaling and ease of deployment, development- are being diminished. You can achieve this with containers.

Factor 10: Dev-Prod Parity

Keep development, staging, and production as similar as possible so anyone can understand it and release it. Continuous deployment needs continuous integration based on matching environments to limit deviation and errors. This also implicitly encourages a DevOps culture where Software Development and Operations are unified. Containerization is a huge help here.

Factor 11: Logs

Treat logs as event streams. Logging is important for debugging and checking up on the general health of your application. At the same time, your application shouldn’t concern itself with the storage of this information. Instead, these logs should be treated as a continuous stream that is captured and stored by a separate service.

Factor 12: Admin Processes

Run admin/management tasks as one-off processes — tasks like database migration or executing one-off scripts in the environment. To avoid messing with the database, use the tooling you built alongside your app to go and check the database.